Verification and Validation of Air Traffic Systems: Tactical Separation Assurance

David Bushnell, Dimitra Giannakopoulou, Peter Mehlitz, Russ Paielli, Corina Păsăreanu *(currently alphabetical – discuss)*
NASA Ames Research Center
Moffett Field, CA 94035, USA
{addresses}@nasa.gov

Abstract—The expected future increase in air traffic requires the development of innovative algorithms and software systems that will automate several functions currently performed by controllers. Extensive verification and validation will be central to ensuring the correct operation of these safety critical systems. Separation assurance, for example, is concerned with maintaining a safe distance between any two aircrafts. It is a complex, real-time problem with many variables and uncertainties, and failure could be disastrous. This paper reports on results from a project at NASA Ames that studies the role of validation and verification technologies in the development of future air-traffic control software systems. It discusses how advanced V&V technologies can contribute to the production of robust software prototypes together with measurable criteria that ensure that the industrial implementations conform to them. It then presents early results obtained from the application of test case generation techniques to the TSAFE component for tactical separation assurance.

TABLE OF CONTENTS?

1. INTRODUCTION

The traffic on U.S. commercial airlines is predicted to increase by the equivalent of two major hub airports each year through 2020. NextGen (Next Generation Air Transportation System) is a NASA research program that addresses this increasing load on the air traffic control system through innovative algorithms and software systems. The seminal work produced by this program should ideally be in the form of reference artifacts that can be adopted by industry. Software verification and validation will be critical to this effort.

This paper discusses a joint project between NextGen and Robust Software Engineering (RSE) groups at NASA Ames that studies the role of validation and verification technologies in the development of future air-traffic control software systems. More specifically, the project aims at the development of techniques and processes that 1) ensure the production of *robust* software prototypes, and 2) provide measurable criteria for checking (automatically) the conformance of industrial implementations to these prototypes.

We will develop techniques that are applicable to several classes of air-traffic control software, and therefore plan on studying several such systems. However, we have focused our initial efforts on testing for the TSAFE (Tactical Separation Assisted Flight Environment) NextGen component. TSAFE is the part of NextGen which seeks to predict and resolve loss of separation in the 30 second to 3 minute time horizon.

Our study of the TSAFE component started by applying off-the-shelf tools to measure test coverage achieved with existing tests applied to TSAFE. The purpose of this step was to assess the complexity of generating appropriate tests for this type of system. The results made it clear that automated support would be crucial in generating tests that would ensure that the component has been tested enough, as well as meaningful regression tests that can be used for checking conformance of industrial implementations to this prototype.

We will discuss our preliminary experiments with automatic test case generation techniques for modules of the TSAFE code. These techniques are implemented in Symbolic Pathfinder (SP), an extension of the Java Pathfinder (JPF) model checking tool. SP adds symbolic execution capabilities to JPF's well developed state space exploration techniques. Through advanced constraint solving modules, SP is able to automatically generate input values that will ensure the traversal of targeted paths in the program. We will also review the challenges that this particular application poses to SP due to its complex numerical computations.

The remainder of the paper is organized as follows. Section 2 is an overview of the TSAFE system, and the verification challenges that it presents. Section 3 discusses the role that we propose for V&V techniques in the development of NextGen algorithms and software systems. It presents symbolic execution as a means of automatically generating high quality software tests, and reviews the Symbolic Pathfinder extension to the JPF. Section 4 describes the application of SPF to TSAFE, our preliminary results and challenges that we have faced. Section 5 summarizes related work, and Section 6 concludes the paper and presents plans for future work.

2. TSAFE

Airspace capacity is the number of flights that can fly safely in a given volume of airspace. In the current air traffic system, that capacity is limited to approximately 15 flights per sector (or controller) due to cognitive limitations of air traffic controllers using radar displays and voice communication with pilots. The large potential increase in air traffic in future decades is expected to require automation of the separation assurance functions that are currently performed by controllers. Separation assurance is a complex, real-time problem with many variables and uncertainties, and failure could be disastrous. The challenge is to develop an automated system that can keep the probability of collision acceptably low, despite the complexity and unpredictability of the traffic patterns, even as traffic doubles or triples.

NASA Ames is developing the Advanced Airspace Concept (AAC) \cite{[AAC],AAC2} to meet that challenge. AAC comprises two stages of separation assurance, plus standard collision avoidance, which constitutes a third stage. The first stage is a strategic auto-resolver \cite{AutoRes} that attempts to detect and resolve conflicts up to approximately 20 minutes in advance. The second stage is a simpler system called the Tactical Separation-Assured Flight Environment (TSAFE), which is intended to backup the strategic auto-resolver and handle any conflicts left undetected or unresolved with loss of separation (LoS) predicted to occur within approximately two minutes. If TSAFE fails to resolve a conflict, the Traffic Alert and Collision Avoidance System (TCAS) \cite{TCAS} is available on most commercial aircraft to prevent a collision using vertical maneuvers.

Because automated conflict detection and resolution are considered safety critical, TSAFE is intentionally designed to be as simple as possible, while still capable of resolving conflicts with high reliability. Thus, TSAFE generates relatively simple maneuvers consisting of altitude, heading, or speed changes, which could be used as controller advisories, but are intended ultimately to be automatically uplinked to the flight deck. For simplicity, TSAFE does not attempt to return the maneuvered flights back to their planned routes after the conflict passes. Because the conflicts are imminent, however, maneuver delays and flight dynamics must be accounted for. TSAFE must also guarantee that conflicts are resolved without creating new conflicts with nearby traffic. In addition, TSAFE must be designed to interact safely with TCAS.

In addition to the far-term objective of tactical conflict resolution, TSAFE has also been developed and tested for a near-term application as a tactical conflict alerting aid for controllers \cite{TSAFE,TSAFE2}. The objective is to eventually replace Conflict Alert, the legacy system for alerting controllers to imminent conflicts in the US. Like Conflict Alert, TSAFE uses constant-velocity (``dead-reckoning'') state projections, but unlike Conflict Alert, it also uses intent information in the form of the flightplan route and the assigned altitude. TSAFE has been tested extensively with actual air traffic data, including archived tracking data for 100 operational errors (losses of separation officially attributed to controller error). TSAFE provided timely alerts more consistently than Conflict Alert. TSAFE was also found to produce substantially fewer false alerts than Conflict Alert.

The tests of TSAFE using archived operational errors and other traffic scenarios go a long way toward establishing credibility in an operational environment, but more testing is needed to guarantee the reliability of TSAFE in actual operation. In particular, traffic scenarios and encounter geometries need to be generated to test any or all ``corner'' cases that could arise in practice. [more?]

TSAFE Interface

The basic inputs and outputs of TSAFE are shown in Fig. \ref{fig:TSAFE-IO}. The primary inputs are radar tracking data and barometric altitude reports from each flight via the standard Mode C datalink. The assigned altitude of each flight is also available to TSAFE via altitude amendments entered by the controller, and the flightplan routes are available through the route amendments entered by the controller. The aircraft type and other basic information for each flight are also provided to TSAFE through basic flight registration. TSAFE can operate without aircraft type information, but it's climb and descent predictions will not be as accurate. Current wind data can also be provided, if available. The output of the near-term version of TSAFE is the predicted conflicts, whereas the output of the far-term version with automated conflict resolution is the recommended resolution maneuvers.

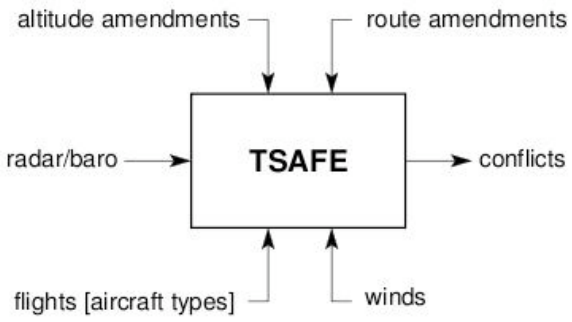

Figure 1: TSAFE Inputs and outputs

The inputs and outputs of TSAFE are provided by function calls. For file replay, each line of text in the input file is a data record that triggers a call to the corresponding input function. The functions interfaces and data records are specified in detail in the TSAFE interface control document \cite{ICD}. The input function names and the corresponding

data record codes are shown in Table \ref{tbl:TSAFE}. Each function call applies to one particular flight, which is specified by the flight identification argument. All of the arguments of the input functions are basic data types such as integers, floating point numbers, and character strings.

Function	**Function Name**	**Code**
flight registration	registerFlight	FLT
route amendment	amendRoute	RTE
altitude amendment	amendAltitude	ALT
radar track update	radarTrack	TRK
flight deletion	deleteFlight	DEL
wind data update	loadWindFile	WND

Table 1: TSAFE input function names and corresponding data record type codes

TSAFE Architecture

TSAFE is implemented as a single instance of a ``TSAFE'' class that is used to manage the entire process. The TSAFE object contains a list of instances of the ``Flight'' class, one for each actual flight being tracked. Each Flight object, in turn, contains an instance of the ``State'' class and the ``Flightplan'' class, each of which will be explained shortly. The TSAFE object has methods for predicting trajectories and checking for conflicts. To account for uncertain pilot intent, TSAFE generates both constant-velocity (state-based) and flightplan-based trajectory predictions for each flight, and it checks all combinations for conflicts.

The ``State'' class represents the dynamic state of a flight in terms of a time-tagged position and velocity. The State class provides a method for computing the ``dead-reckoning'' (constant-velocity) horizontal path and altitude predictions. The predictions are stored as a series of points at regular, synchronous intervals of time (default: 6 seconds). The state class also contains many utility functions for various basic tasks such as time projection and synchronization, computation of current separation and predicted minimum separation at constant velocity.

The ``Flightplan'' class represents the planned route and performs computations associated with that route. It takes a list of two-dimensional route waypoints and constructs a nominal path with tangent turn arcs of a constant radius based on a coordinated turn at a specified bank angle (default: 20 degrees). The resulting route consists of an alternating sequence of straight and turn segments, each type represented as an instance of a small utility class. The Flightplan class has a function that accepts the current state (position and velocity) of the flight and constructs a predicted trajectory that converges to the rounded path at a specified convergence angle (default: 10 degrees). It also has a function to compute the cross-track and course error of a given state. Those errors are used to determine the prediction look-ahead time of the state-based (constant-velocity) and flightplan-based trajectories. The closer the flight is following its flightplan, the longer is the prediction time for the flightplan-based trajectory prediction and the shorter is the state-based prediction.

The ``Flight'' class represents a single flight. Each Flight object contains a State object and a Flightplan object. It uses the State object to generate the constant-velocity predictions, and it uses the Flightplan object to generate the flightplan-based predictions. The Flightplan class also has a method for checking for conflict between a pair of flights. That method is called by the TSAFE object for each pair of flights. The Flight class uses the ACmodels class to compute predicted climb and descent rates in response to altitude amendments entered by the controller.

To check for conflicts between a pair of flights, the Flight class uses the ``PotentialTrajectories'' class. The PotentialTrajectories class accepts a list of horizontal trajectory predictions and a list of vertical predictions for each flight. The lists currently consist of the constant-velocity predictions and the flightplan-based predictions, but other types of predicted trajectories could be added later if necessary. The vertical predictions are reduced to an altitude range at each point in predicted time and superimposed over both of the horizontal trajectories. All four combinations of trajectories are then checked for conflict. An trajectory bounding procedure is used to eliminate the detailed checking for conflicts if the two flights are far apart.

An experimental class that can optionally be used by each Flight object is the ``TurnDetection'' class. If a flight is not following its planned route, this class analyzes the tracking history and attempts to detect the initiation of an unplanned turn. An unplanned turn is a turn for which the controller enters no data into the Host computer. Such turns occur in holding patterns, for example. They also occur when a flight is turned to resolve a conflict, or when a flight is turned to get it back on its planned route after a conflict has been resolved. When an unplanned turn is detected, a predicted trajectory is constructed in which the turn continues for some angle. The challenge here is to avoid false alerts due to noise in the radar tracking data, which can occasionally be severe. When improved surveillance methods become available in the future, this class should be more effective and produce fewer false alerts.

In the interest of brevity, several other utility classes will not be discussed here. Also, the architecture for conflict resolution will not be discussed.

3. TEST CASE GENERATION

Test Case Generation for NextGen

(add picture from slides and extend this text)

1. Provide synthetic test data and test oracles that will thoroughly and automatically verify TSAFE,

2. Provide support for extended notions of coverage, and

3. Reduce the size of the test suite so that it retains the necessary coverage while focusing on the most interesting and meaningful tests.

For test case generation we use Symbolic PathFinder \cite{SPF}, a recent extension to JPF that combines symbolic execution and constraint solving for automated test case generation. Symbolic PathFinder implements a symbolic execution framework for Java byte-code. It can handle mixed integer and real inputs, as well as multi-threading and input pre-conditions. We describe symbolic execution and the Symbolic PathFinder tool in more detail below.

Symbolic Execution

Symbolic execution \cite{King76SymbolicExecutionProgramTesting} is a form of program analysis that uses symbolic values instead of actual data as inputs and symbolic expressions to represent the values of program variables. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a path condition (PC), and a program counter. The path condition is a boolean formula over the symbolic inputs, encoding the constraints which the inputs must satisfy in order for an execution to follow the particular associated path. These conditions can be solved (using off-the-shelf constraint solvers) to generate test cases (test input and expected output pairs) guaranteed to exercise the analyzed code. The paths followed during the symbolic execution of a program are characterized by a *symbolic execution tree*.

```
\begin{figure}
\begin{verbatim}
[1]  if ((pressure < pressure_min) ||
[2]      (pressure > pressure_max)) {
[3]    ... /* abort */
       } else {
[4]    ... /* continue */
       }
\end{verbatim}
\caption{Example for symbolic execution.}
\label{fig:symexe}
\end{figure}
```

To illustrate the difference between concrete and symbolic execution, consider the example in Figure~\ref{fig:symexe}. The code checks if the value of pressure
(input variable $pressure$) is within min and max allowed values (input variables $pressure\_min$ and $pressure\_max$). In concrete execution (e.g. testing) one executes the code on given concrete inputs. For example, for $pressure=460$, $pressure\_min=640$, $pressure\_max=960$, only one path through the code will be executed, corresponding to the first disjunct in the {\tt if} statement being true. In contrast, symbolic execution starts with symbolic input values ($pressure=Sym1$, $pressure\_min = MIN$ and $pressure\_max=MAX$). Symbolic execution will analyze three paths through the program and it will generate three path conditions, according to different possibilities in the code:

\noindent $PC_1: Sym_1 < MIN$,\\
$PC_2: Sym_1 > MAX$,\\
$PC_3: Sym_1 \geq MIN \wedge Sym_2 \leq MAX$

Concrete values for the inputs that satisfy ("solve") the path conditions are then found with the help of a constraint solver and those solutions are used as concrete test inputs that are guaranteed to give full path coverage for this code.

Symbolic PathFinder

Symbolic PathFinder implements a non-standard interpreter for byte-codes on top of JPF. The symbolic information is stored in attributes associated with the program data and it is propagated on demand, during symbolic execution. The analysis engine of JPF is used to systematically generate and explore the symbolic execution tree of the program. JPF is also used to systematically analyze thread interleavings and any other forms of non-determinism that might be present in the code; furthermore JPF is used to check properties of the code during symbolic execution. Off-the-shelf constraint solvers/decision procedures {\tt choco} and {\tt IASolver} \cite{DP} are used to solve mixed integer and real constraints. We handle loops by putting a bound on the model-checker search depth and/or on the number of constraints in the path conditions.

4. CASE STUDY: APPLICATION OF TEST GENERATION/SPF TO TSAFE

As was described in Section 1, the prototype for the TSAFE algorithms has been written in an object-oriented style with the TSAFE class managing the calculations and various other classes implementing key parts of the algorithms.

The TSAFE prototype is written in the Python programming language. Python is a widely used scripting language that makes prototyping quick and easy.

During its development, TSAFE has been tested with data derived from real air traffic control data. Being derived from real data, the test data has the virtue of accurately depicting real aircraft operations. However, because loss of separation (LOS) occurs very rarely in practice, it only tests a limited number of LOS scenarios and limited parts of the code.

We verified this by using a standard off-the-shelf Python coverage tool, coverage.py [PCC], to measure actual code coverage under the real-data tests. coverage.py, like most commonly available code coverage tools, measures only statement coverage. It does not compute more sophisticated coverage statistics such as MC/DC coverage or path coverage.

The results are shown in Table 4.1. The code coverage from the tests derived from the real data ranged from poor to good. In one case, the TrackFilter class, 99% of the statements were executed and in several other cases, more than 85% coverage was attained. But in many classes the coverage was spotty at best.

Table 4.1 Code Coverage for Real Test Data

Class	*Statements*	*Executed*	*Coverage*
ACmodels	189	163	86%
Flight	465	413	88%
FlightRes	197	20	10%
Flightplan	332	220	66%
RingBuffer	50	26	52%
State	255	135	52%
TSAFE	103	80	77%
TSAFEreplay	228	166	72%
TSAFEres	127	20	15%
TrackFilter	115	114	99%
Trajectory	351	308	87%
Turnlogic	204	17	8%

Since the prototype is still under development, this was to be expected for some of the classes. (There is newly written code that has not yet been tied in to the rest of the system.) But even allowing for this, it is clear that more extensive test suites are needed. We therefore applied SPF's model checking and symbolic execution capabilities to the TSAFE code.

Since SPF is a model checker for Java bytecode, the Python code was first translated to Java. This process was straightforward -- since the Python code was well structured, it was possible to do a simple, almost mechanical translation. While doing the translation, we were careful to preserve the logical structure of the original code so that tests generated from the Java code would be meaningful to the Python code.

Even though the TSAFE prototype is well structured, it is still a complex program: it necessarily uses many complex data types, has many loops and complex logic, and uses floating point arithmetic extensively. All of these present challenges to model checkers.

In our initial investigations, we have therefore taken a bottom-up approach, applying SPF to the lowest level routines. This allows us to concentrate our initial work on addressing the issues of model checking floating point code and complex data structures.

Once the SPF extensions for model checking and symbolic execution of floating point code described in section 3 were added, we applied SPF to the low level methods aircraft_turn_status in the class TurnLogic and currentLOS in the class “Flight”. These methods were described in Section 1. Recall that aircraft_turn_status determines whether an aircraft should be turning left, right, or going straight. currentLOS determines whether two aircraft have a loss of separation.

These methods were chosen because (1) they used non-trivial floating point operations, (2) they contained significant program logic (for example, multiple nested if statements), and (3) their behavior depended on both the methods' input parameters and their objects' internal state.

The methods required some modification before they could be run under SPF. In the case of aircraft_turn_status, the original method returned a string as its result and depended on a string instance variable. Since SPF does not yet handle strings, we converted these parameters and variables to integers. This worked well because the strings only represented three distinct values, LEFT, RIGHT, and STRAIGHT. Some of the code from this method is shown in Figure 4.1.

Figure 4.1: aircraft_turn_status code

```
_initial_turn_active = false;

if (STRAIGHT == _turnStatus) {
  turn_status = _activate_ts(p0, p1, p2);
  _initial_turn_active = true;

} else if (RIGHT == _turnStatus) {
  double phi_stop
    = _angle_to_stop_turn(_turnStatus, p1);

  if (p0 > bank_turn_crit_next)
    turn_status = RIGHT;

  else if (p0 > 0.0 && p01 > 40.0)
```

SPF generated approximately 350 test cases for aircraft_turn_status. These tests will exercise all paths though the method. If a less stringent coverage metric is required (e.g. statement coverage or branch coverage), then the run could be tuned and would produce fewer test cases. Some of the test cases generated by SPF are shown in Table 4.2. These 350+ test cases took about two minutes to generate.

Test Case	p0	p1	p2	turnStatus	Return Value
1	-5008.0	-5006.5	don't care	Straight	Left
2	-5016.0	-13.0	-5006.5	Straight	Left
3	-5016.0	-13.0	-13.0	Straight	Straight
4	-7508.0	-13.0	4971.0	Straight	Straight
. . .					
347	-5005.0	4995.0	don't care	Left	Left
348	-10.0	4995.0	don't care	Left	Straight
349	-5.0	4995.0	don't care	Left	Straight
350	0.0	4995.0	don't care	Left	Straight
351	5000.0	4995.0	don't care	Left	Straight

The method currentLOS required somewhat more extensive changes, but this time the changes were required because of limitations in the constraint solver. (Recall that any given constraint solver over real numbers will have limitations because the general constraint solving problem for reals is undecidable.) In our case the constraint solver cannot handle constraints involving square roots.

currentLOS uses square roots because it must compute the distance between two points. However, the method only uses that distance to do comparisons with other numbers. So the square root can be eliminated by using the square of the distance in place of the distance.

Once the changes were made, we were able to run SPF on currentLOS and produce our suite of test cases for all paths through the code. However, the logic in currentLOS is fairly complex and requiring complete coverage of all possible paths produces well over one hundred megabytes of test cases.

The results of automatic test generation for the currentLOS method illustrate the point that full path coverage is often not a practical requirement for testing, even for systems requiring high reliability. In practice, a less stringent form of code coverage is required. As we scale up to larger and more complex methods we will necessarily need to tune SPF for less demanding coverage metrics.

5. Related Work

Work on symbolic execution.

Nancy Levenson's work on TSAFE.

Work by Tevfik Bultan and Others on TSAFE.

Work focusing on algorithms by Langley partners.

6. Conclusions & Future Work

7. ACKNOWLEDGMENTS

REFERENCES

[AAC]

\bibitem{AAC} Erzberger, H.: ``The Automated Airspace Concept," 4th USA/Europe Air Traffic Management R\&D Seminar, Santa Fe, NM, USA, 3--7 Dec. 2001.

\bibitem{AAC2} Erzberger, H.; Paielli, R.A.: ``Concept for Next Generation Air Traffic Control System," \emph{Air Traffic Control Quarterly}, Vol. 10(4)(2002), pp 355-378.

\bibitem{AutoRes} Erzberger, H.: ``Automated Conflict Resolution for Air Traffic Control," 25th International Congress of the Aeronautical Sciences (ICAS), Hamburg, Germany, 3--8 Sep. 2006.

\bibitem{TCAS} Introduction to TCAS II Version 7. Federal Aviation Administration, Nov. 2000.

\bibitem{TSAFE} Paielli, R.A.; Erzberger, H.: ``Tactical Conflict Detection Methods for Reducing Operational Errors," \emph{Air Traffic Control Quarterly}, Vol. 13(1)(2005).

\bibitem{TSAFE2} Paielli, R.A.; Erzberger, H., Chiu, D., and Heere, K.R: ``Tactical Conflict Alerting Aid for Air Traffic Controllers," accepted for publication in the AIAA \emph{Journal of Guidance, Control, and Dynamics}, 2008.

\bibitem{ICD} Paielli, R.A.; ``TSAFE Interface Control Document."

\bibitem{BADA} Eurocontrol: \emph{User Manual for the Base of Aircraft Data (BADA)}, Revision 3.6, EEC Note No. 10/04, ACE-C-E2, Eurocontrol Experimental Centre, July 2004.

BIOGRAPHY

Ed Bryan *is a consultant in software development methodology and technology. He has developed and led development of software at Bell Labs, RAND, Scientific Data Systems, Xerox, Honeywell, and Groupe Bull. He previously served as Director of Honeywell and Groupe Bull's Los Angeles Development Center, where the operating systems, databases, compilers, and communications software for the CP-6 system on main-frame hardware were developed and supported. He has held management positions at startups International Meta Systems in 1987 and Acorn Technologies in 1997. In the early 1960s he wrote the operating system for JOSS II, one of the earliest timesharing systems. He has a BSEE from Caltech and a master's certificate in communications from Bell Laboratorie*